CSE - 584 Machine Learning

Homework - 2

Reinforcement Learning (RL) is an area of machine learning where an agent learns to make decisions by interacting with an environment and receiving feedback in the form of rewards. One of the most fundamental RL algorithms is **Q-learning**, a model-free algorithm that seeks to learn an optimal action-selection policy for a given environment.

In this project, we explore the application of Q-learning in the **FrozenLake** environment, a classic RL problem. The task involves navigating a 4x4 grid, where the agent must reach a goal while avoiding obstacles (holes) scattered across the environment. The agent interacts with the environment by selecting actions (up, down, left, right), updating its knowledge through the Q-table, which holds the expected future rewards for each state-action pair.

This implementation is inspired by the GitHub repository **Barboss4/Q-learning-Frozenlake**, where Q-learning is employed to help the agent successfully traverse the frozen lake. The algorithm balances exploration and exploitation through an epsilon-greedy policy and progressively refines the Q-table to maximise cumulative rewards, demonstrating effective decision-making in stochastic environments.

For reference to the GitHub project: Barboss4/Q-learning-FrozenlakeGitHub

1. Writing an abstract that provides a high - level overview of the code's purpose and the overall process it follows.

This code implements a Q-Learning reinforcement learning algorithm to solve the FrozenLake problem, a classic environment from OpenAI's Gym library. The objective of the agent is to navigate through a 4x4 grid where each tile represents either frozen ground (safe) or a hole (dangerous), and reach a designated goal without falling into holes. By leveraging a Q-learning approach, the agent learns an optimal policy through interaction with the environment, updating a Q-table that maps states to actions with the highest expected future rewards.

The code follows a structured process, beginning with initializing the environment and creating the Q-table for tracking state-action values. It uses an epsilon-greedy strategy to balance exploration (trying new actions) and exploitation (choosing the best-known action) during the agent's learning process. The Q-values are updated iteratively using the Bellman equation, which refines the agent's understanding of optimal actions based on the received rewards. After training the agent for several episodes, the learned policy is evaluated by running the agent through multiple trials, followed by rendering the agent's performance in the form of a GIF.

The use of dynamic exploration rates, well-structured evaluation, and visual output ensures a robust implementation that allows the agent to learn efficiently while providing a clear depiction of the policy's effectiveness. This implementation showcases how reinforcement learning can be applied to real-world-like navigation problems, demonstrating the agent's ability to learn an optimal path in uncertain environments.

2. Identify the core section (such as a couple of functions about the RL implementations) of the reinforcement learning implementation and add comments to each line, explaining what it is doing. This demonstrates by your step by step understanding.

Core section from the reinforcement learning implementation with detailed comments to demonstrate understanding::

```
def train(n_training_episodes, min_epsilon, max_epsilon, decay_rate, env, max_steps, Qtable):

  # Loop over the number of training episodes

  for episode in trange(n_training_episodes):


    # Dynamically calculate epsilon for the current episode, decaying it over time to reduce exploration

    epsilon = min_epsilon + (max_epsilon - min_epsilon) * np.exp(-decay_rate * episode)


    # Reset the environment to the initial state at the start of each episode

    state = env.reset()

    step = 0

    done = False


    # Loop for the maximum number of steps in each episode

    for step in range(max_steps):


      # Select an action using the epsilon-greedy policy

      action = epsilon_greedy_policy(Qtable, state, epsilon)


      # Take the selected action, observe the new state and reward from the environment

      new_state, reward, done, info = env.step(action)
```

```
        # Update the Q-value for the state-action pair using the Bellman equation

        # Q(s, a) = Q(s, a) + learning_rate * [reward + gamma * max(Q(s', a')) - Q(s, a)]

        Qtable[state][action] = Qtable[state][action] + learning_rate * (reward + gamma * np.max(Qtable[new_state]) - Qtable[state][action])


        # If the episode ends (goal reached or falls into a hole), break the loop

        if done:

            break


        # Update the current state to the new state for the next step

        state = new_state


    # Return the updated Q-table after completing all episodes

    return Qtable
```

Training Loop (`for episode in trange(n_training_episodes)`):

- This loop iterates over the number of training episodes. Each episode represents a complete interaction with the environment from start to termination (either by reaching the goal or falling into a hole).

Dynamic Epsilon Calculation:

- Epsilon is decayed using the formula `epsilon = min_epsilon + (max_epsilon - min_epsilon) * np.exp(-decay_rate * episode)`, allowing the algorithm to shift from exploration (random actions) to exploitation (choosing the best action based on the Q-table) as training progresses.

Environment Reset:

- The `state = env.reset()` command initializes the environment to a starting state for a new episode.

Epsilon-Greedy Action Selection:

- The action is selected using an epsilon-greedy strategy, which allows the agent to explore random actions with probability epsilon or exploit the best-known action from the Q-table with probability (1 - epsilon).

Environment Interaction:

- The chosen action is executed by the environment using `env.step(action)`, which returns:
  - The next state (`new_state`)
  - The reward for the action (`reward`)
  - A flag indicating if the episode has ended (`done`)

Q-Value Update:

- The Q-value for the current state-action pair is updated using the Bellman equation: Q(s,a)=Q(s,a)+α[r+γmaxQ(s′,a′)−Q(s,a)]Q(s, a) = Q(s, a) + \alpha \left[r + \gamma \max Q(s', a') - Q(s, a) \right]Q(s,a)=Q(s,a)+α[r+γmaxQ(s′,a′)−Q(s,a)] This formula updates the expected value of the action by factoring in the immediate reward and the maximum expected reward from the next state.

Termination:

- The loop terminates early if the episode is completed (i.e., the agent has reached the goal or fallen into a hole).

Return Updated Q-table:

- After training over all episodes, the updated Q-table is returned, which will be used in the evaluation phase.